Distributed Databases

Chapter 22, Part B

Introduction

- Data is stored at several sites, each managed by a DBMS that can run independently.
- Distributed Data Independence: Users should not have to know where data is located (extends Physical and Logical Data Independence principles).
- Distributed Transaction Atomicity: Users should be able to write Xacts accessing multiple sites just like local Xacts.

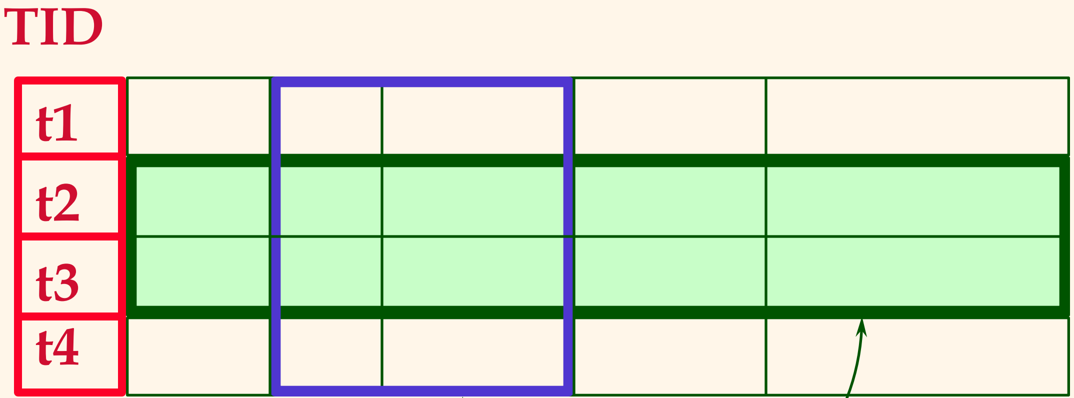
Recent Trends

- Users have to be aware of where data is located, i.e., Distributed Data Independence and Distributed Transaction Atomicity are not supported.
- These properties are hard to support efficiently.
- For globally distributed sites, these properties may not even be desirable due to administrative overheads of making location of data transparent.

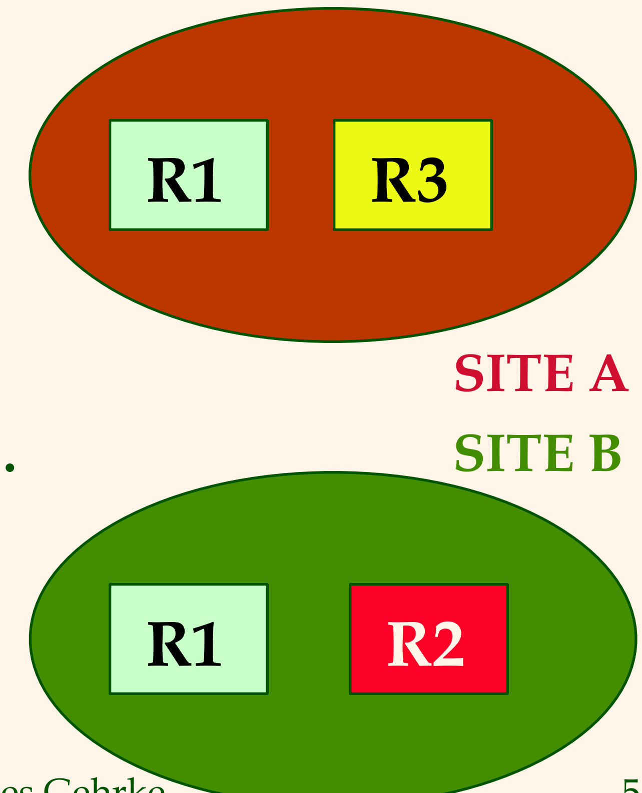
Types of Distributed Databases

- **Homogeneous:** Every site runs same type of DBMS.
- **Heterogeneous:** Different sites run different DBMSs (different RDBMSs or even non-relational DBMSs).

Storing Data



- Fragmentation
  - Horizontal: Usually disjoint.
  - Vertical: Lossless-join; tids.
- Replication
  - Gives increased availability.
  - Faster query evaluation.
  - Synchronous vs. Asynchronous.
    - Vary in how current copies are.



Distributed Catalog Management

- Must keep track of how data is distributed across sites.
- Must be able to name each replica of each fragment. To preserve local autonomy:
  - <**local-name**, **birth-site**>
- Site Catalog: Describes all objects (fragments, replicas) at a site + Keeps track of replicas of relations created at this site.
  - To find a relation, look up its birth-site catalog.
  - Birth-site never changes, even if relation is moved.

Distributed Queries

```
SELECT AVG(S.age)
FROM Sailors S
WHERE S.rating > 3
    AND S.rating < 7
```

- Horizontally Fragmented: Tuples with rating < 5 at Shanghai, >= 5 at Tokyo.
  - Must compute SUM(age), COUNT(age) at both sites.
  - If WHERE contained just S.rating>6, just one site.
- Vertically Fragmented: *sid* and *rating* at Shanghai, *sname* and *age* at Tokyo, *tid* at both.
  - Must reconstruct relation by join on *tid*, then evaluate the query.
- Replicated: Sailors copies at both sites.
  - Choice of site based on local costs, shipping costs.

Distributed Joins

LONDON	PARIS
Sailors	Reserves
500 pages	1000 pages

- Fetch as Needed, Page NL, Sailors as outer:
  - **Cost:** 500 D + 500 * 1000 (D+S)
  - **D** is cost to read/write page; **S** is cost to ship page.
  - If query was not submitted at London, must add cost of shipping result to query site.
  - Can also do INL at London, fetching matching Reserves tuples to London as needed.
- Ship to One Site: Ship Reserves to London.
  - Cost: 1000 S + 4500 D (SM Join; cost = 3*(500+1000))
  - If result size is very large, may be better to ship both relations to result site and then join them!

Semijoin

- At London, project Sailors onto join columns and ship this to Paris.
- At Paris, join Sailors projection with Reserves.
  - Result is called **reduction** of Reserves wrt Sailors.
- Ship reduction of Reserves to London.
- At London, join Sailors with reduction of Reserves.
- Idea: Tradeoff the cost of computing and shipping projection and computing and shipping projection for cost of shipping full Reserves relation.
- Especially useful if there is a selection on Sailors, and answer desired at London.

Bloomjoin

- At London, compute a bit-vector of some size k:
  - Hash join column values into range 0 to k-1.
  - If some tuple hashes to I, set bit I to 1 (I from 0 to k-1).
  - Ship bit-vector to Paris.
- At Paris, hash each tuple of Reserves similarly, and discard tuples that hash to 0 in Sailors bit-vector.
  - Result is called **reduction** of Reserves wrt Sailors.
- Ship bit-vector reduced Reserves to London.
- At London, join Sailors with reduced Reserves.
- Bit-vector cheaper to ship, almost as effective.

Distributed Query Optimization

- Cost-based approach; consider all plans, pick cheapest; similar to centralized optimization.
  - Difference 1: Communication costs must be considered.
  - Difference 2: Local site autonomy must be respected.
  - Difference 3: New distributed join methods.
- Query site constructs global plan, with suggested local plans describing processing at each site.
  - If a site can improve suggested local plan, free to do so.

Updating Distributed Data

- Synchronous Replication: All copies of a modified relation (fragment) must be updated before the modifying Xact commits.
  - Data distribution is made transparent to users.
- Asynchronous Replication: Copies of a modified relation are only periodically updated; different copies may get out of synch in the meantime.
  - Users must be aware of data distribution.

Synchronous Replication

- Voting: Xact must write a majority of copies to modify an object; must read enough copies to be sure of seeing at least one most recent copy.
  - E.g., 10 copies; 7 written for update; 4 copies read.
  - Each copy has version number.
  - Not attractive usually because reads are common.
- Read-any Write-all: Writes are slower and reads are faster, relative to Voting.
  - Most common approach to synchronous replication.
- Choice of technique determines *which* locks to set.

Cost of Synchronous Replication

- Before an update Xact can commit, it must obtain locks on all modified copies.
  - Sends lock requests to remote sites, and while waiting for the response, holds on to other locks!
  - If sites or links fail, Xact cannot commit until they are back up.
  - Even if there is no failure, committing must follow an expensive ***commit protocol*** with many msgs.
- So the alternative of *asynchronous replication* is becoming widely used.

Asynchronous Replication

- Allows modifying Xact to commit before all copies have been changed (and readers nonetheless look at just one copy).
  - Users must be aware of which copy they are reading, and that copies may be out-of-sync for short periods of time.
- Two approaches: Primary Site and Peer-to-Peer replication.
  - Difference lies in how many copies are ``updatable'' or ``master copies''.

Peer-to-Peer Replication

- More than one of the copies of an object can be a master in this approach.
- Changes to a master copy must be propagated to other copies somehow.
- If two master copies are changed in a conflicting manner, this must be resolved. (e.g., Site 1: Joe's age changed to 35; Site 2: to 36)
- Best used when conflicts do not arise:
  - E.g., Each master site owns a disjoint fragment.
  - E.g., Updating rights owned by one master at a time.

Primary Site Replication

- Exactly one copy of a relation is designated the primary or master copy. Replicas at other sites cannot be directly updated.
  - The primary copy is published.
  - Other sites subscribe to (fragments of) this relation; these are secondary copies.
- Main issue: How are changes to the primary copy propagated to the secondary copies?
  - Done in two steps. First, capture changes made by committed Xacts; then apply these changes.

Implementing the Capture Step

- **Log-Based Capture:** The log (kept for recovery) is used to generate a Change Data Table (CDT).
  - If this is done when the log tail is written to disk, must somehow remove changes due to subsequently aborted Xacts.
- **Procedural Capture:** A procedure that is automatically invoked (trigger; more later!) does the capture; typically, just takes a snapshot.
- Log-Based Capture is better (cheaper, faster) but relies on proprietary log details.

Implementing the Apply Step

- The Apply process at the secondary site periodically obtains (a snapshot or) changes to the CDT table from the primary site, and updates the copy.
  - Period can be timer-based or user/application defined.
- Log-Based Capture plus continuous Apply minimizes delay in propagating changes.
- Procedural Capture plus application-driven Apply is the most flexible way to process changes.

Data Warehousing and Replication

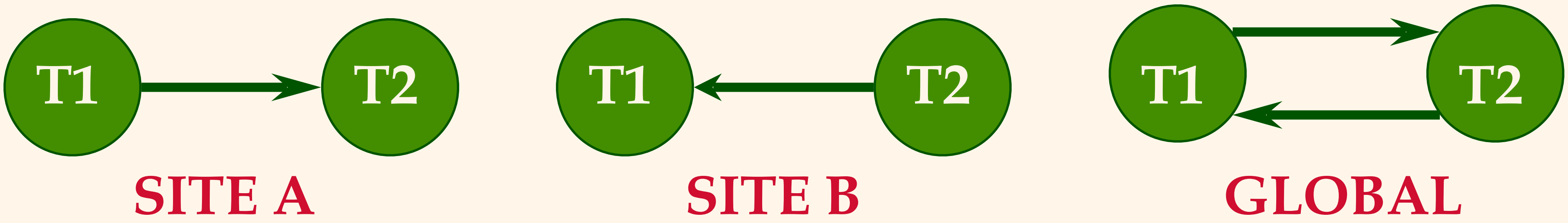
- A hot trend: Building giant "warehouses" of data from many sites.
  - Enables complex decision support queries over data from across an organization.
- Warehouses can be seen as an instance of asynchronous replication.
  - Source data typically controlled by different DBMSs; emphasis on "cleaning" data and removing mismatches ($ vs. rupees) while creating replicas.
- Procedural capture and application Apply best for this environment.

Distributed Locking

- How do we manage locks for objects across many sites?
  - Centralized: One site does all locking.
    - Vulnerable to single site failure.
  - Primary Copy: All locking for an object done at the primary copy site for this object.
    - Reading requires access to locking site as well as site where the object is stored.
  - Fully Distributed: Locking for a copy done at site where the copy is stored.
    - Locks at all sites while writing an object.

Distributed Deadlock Detection

- Each site maintains a local waits-for graph.
- A global deadlock might exist even if the local graphs contain no cycles:



- Three solutions: Centralized (send all local graphs to one site); Hierarchical (organize sites into a hierarchy and send local graphs to parent in the hierarchy); Timeout (abort Xact if it waits too long).

Distributed Recovery (A & D)

- Will discuss this at a high level here
  - Highly recommended to read Phil Bernstein's book, Chapter 7 for a more thorough presentation
- Even in the absence of failures, need a commit protocol to make sure everyone agrees whether to abort or commit
- Should also allow ``cleanup'' after a failure to reach a consistent state

Failures in a distributed system

- Site failures: site crashes, processing stops and the contents of RAM disappear
  - But assume will not fail in another way, e.g. by sending incorrect messages
  - Either up and working correctly or completely down

Failures in a distributed system

- Site will eventually come back up and is able to run some recovery protocol
- Site will keep a log and force the log to disk at particular points. Log will survive crash

Failures in a distributed system

- Communication failures: site A may become unable to reach site B
- Could be caused by a network problem, in particular a network partition
- As links recover, communication will be reestablished
- Messages that are undeliverable are dropped
  - If A receives no reply from B, cannot tell whether B is down or B is up but have a comm failure

Detecting failures

- Failures can be detected by timeouts
- If A contacts B and doesn’t get a reply within a predetermined timeout period, concludes that it cannot communicate with B
- Timeout period determined empirically based on system properties

Consensus Protocols

- A number of consensus protocols exist to allow sites in a distributed system to agree on something
  - e.g. whether to commit
  - We discuss a few of them next
- Start with Two-Phase Commit (2PC)
  - Not to be confused with 2PL ☺
- May also see the term XA transactions
  - Industry standard/specification for distributed transactions
  - A variant of 2PC

Two Phase Commit

- Every node is either a:
  - Coordinator, or
  - Subordinate
- May have different coordinators for different transactions

Two Phase Commit

- Two rounds of communication (hence two phase)
  - Voting phase
  - Termination phase

Phase 1 (Voting)

- Coordinator sends prepare message to each subordinate
- Each subordinate
  - Decides what to do (commit or abort)
  - Communicates decision to coordinator (yes or no)

Phase 2 (Termination)

- Coordinator receives yes/no messages from all subordinates
  - If all are yes, transaction will commit
    - Sends commit message to subordinates
  - If at least one is no, transaction will abort
    - Sends abort message to subordinates

Phase 2 (Termination), cont.

- Upon receiving commit, a subordinate
  - Commits its portion of the transaction
  - Sends ack to coordinator
- Symmetrically upon receiving abort

Two-Phase Commit

Coordinator

Send prepare

Subordinate

Make local decision

Send yes or no

Wait for all responses

Decide abort or commit

Send abort or commit

Perform abort or commit

Send ACK

Wait for all ACKs